CA (MI190) - cours 5 : contrôle de haut niveau

signaux, exceptions, continuations

Emmanuel Chailloux

Plan du cours

Contrôle de haut niveau:

- Signaux : ruptures de calcul et reprise immédiate
- Exceptions : ruptures de calcul
- Continuation : rupture ET reprise de calcul (GOTO fonctionnel)

Ruptures de calcul : signaux

signaux en C:

- Les signaux en C permettent d'informer un processus qu'un événement particulier s'est produit.
- Chaque processus peut gérer un traitement particulier pour un signal donné.
- Le traitement lié à un signal associe à un signal une fonction de traitement.
- Lors du déclenchement de cet événement le programme exécute la fonction associée puis reprend l'exécution du programme là où il l'avait laissé.

différences importantes entre BSD et SYSTEM 5,
voir le fichier `<signal.h>`

Exemple 1

- Le signal `SIGINT` indique une interruption (`kill -2` ou `^C`) du processus. Poser un récupérateur pour le signal `SIGINT` qui affiche le message `"Pas d'interruption possible"` en cas d'arrivée de ce signal.

```
#include <signal.h>
void hand_int(int sig){ printf("Pas d'interruption ^C active\n");
  printf("Taper ^\\ si vous d\'esirez sortir du programme\n");
  signal(SIGINT,hand_int);
}
main () { int i;
  int c=0;
  signal(SIGINT,hand_int);
  printf("un programme qui boucle, essayer ^C pour l'arreter\n");
  while (1)
    {i++;
     if ((i % 100000) == 0) {c++;printf("."); fflush(stdout);}
    }
}
```

Exemple 2 - (1)

```
#include <signal.h>
#include <stdio.h>

struct cons {int car; struct cons *cdr;};
typedef struct cons *liste_entier;

liste_entier cons(int a, liste_entier l){
 liste_entier r;
 r=(liste_entier)(malloc(sizeof(struct cons)));
 r->car=a; r->cdr=l; return r;
}

liste_entier intervalle(int a, int b){
  if (a > b) return NULL;
  else return cons(a,intervalle(a+1,b));
}

int compteur=0;
```

Signaux : exemple 2 - (2)

```
int compte(liste_entier l){
  sleep(1);
  if (l == NULL) return 0;
  else {
    compteur++;
    return 1 + compte(l->cdr);
  }
}
void hand(){
  printf(" deja compt\'es %d\n",compteur);
  signal(SIGINT,hand);
}
main(int argc, char *argv[]){
 int r;
 int x;
 if (argc == 1) fprintf(stderr,"pas d'arguments\n");
 else {
   signal(SIGINT,hand);
   x = atoi(argv[1]);
   r = compte(intervalle(1,x));
   printf("%d elements\n",r);
 }
```

Typage et domaine de définition

type inféré $\neq$ domaine de définition:

- c'est une approximation
- exemple : division entière, tête de liste vide
- provient souvent d'un filtrage non exhaustif

Que faire?:

- utiliser une valeur spéciale

```
# asin 2.;;
- : float =          NaN
```

- effectuer une rupture de calcul jusqu'à un récupérateur (exceptions)

Exceptions

Syntaxe:

Exception E1;; ou **Exception** E1 **of** t1;;

- une exception est une valeur de type *exn*
- le type *exn* est un type somme monomorphe **extensible**

```
# exception A_MOI;;
exception A_MOI
# A_MOI;;
- : exn = A_MOI
# exception Depth of int;;
exception Depth of int
# Depth 4;;
- : exn = Depth(4)
```

Déclenchement d'une exception

`raise` : *exn* `-> 'a`

- impossible à écrire $\Rightarrow$ primitive
- l'expression (`raise` E1) n'a pas de contrainte de type

```
# raise A_MOI;;
Uncaught exception: A_MOI
# let x = 18;;
val x : int = 18
# if (x = 0) then raise A_MOI else x;;
- : int = 18
```

Déclarations et déclenchements d'exception (1)

```
# exception Echec of string;;
exception Echec of string

# let declenche_echec s = raise (Echec s);;
val declenche_echec : string -> 'a = <fun>

# declenche_echec "argument invalide";;
Exception: Echec "argument invalide".
```

la fonction `failwith` s'écrit :

```
let failwith s = raise (Failure s);;
let invalid_argument s =
   raise (Invalid_argument s);;
```

Déclarations et déclenchements d'exception (2)

```
# exception OrthoExn  of int * int * string;;
exception OrthoExn of int * int * string

# raise (OrthoExn (3, 6, "le caml"));;
Exception: OrthoExn (3, 6, "le caml").

# exception FuncTreat of (int -> int);;
exception FuncTreat of (int -> int)

# raise (FuncTreat (fun x -> x + 1));;
Exception: FuncTreat <fun>.
```

Déclarations et déclenchements d'exception (3)

Filtrage de motifs incomplet:

```
# let tete l = match l with t::q -> t;;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val tete : 'a list -> 'a = <fun>
# tete [1;2;3];;
- : int = 1
# tete [];;
Exception: Match_failure ("", 13, 35).
# exception Found_zero;;
exception Found_zero
# let rec mult_aux l= match l with
     h::[] -> h
  |  0::t -> raise Found_zero
  |  h::t -> h * mult_aux t;;
Warning: this pattern-matching ...
val mult_aux : int list -> int = <fun>
```

Récupération d'exceptions

Syntaxe:

try *expr* **with** *filtrage*

Le type des motifs du *filtrage* doit être *exn*.

```
# let mult_list l = match l with
  [] -> 0
| lo ->  try mult_aux lo with
    Found_zero -> 0;;
val mult_list : int list -> int = <fun>

# mult_list [1;2;3;0;5;6];;
- : int = 0
```

Exemples fonctionnels

```
#let rec fold_left f accu l =
  match l with
    [] -> accu
  | a::l -> fold_left f (f accu a) l
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
# let rec fold_right f l accu =
  match l with
    [] -> accu
  | a::l -> f a (fold_right f l accu)
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
# fold_left (/) 1000 [3;5;11];;
- : int = 6
# fold_left (/) 1000 [3;0;11];;
Exception: Division_by_zero.
# let idiv a b = b / a;;
val idiv : int -> int -> int = <fun>
# fold_right idiv [3;5;11]  1000;;
- : int = 6
# fold_right idiv  [3;0;11] 1000;;
Exception: Division_by_zero.
```

Utilisation des exceptions

- Gestion de situations exceptionnelles où le calcul ne peut pas se poursuivre $\rightarrow$ rupture du calcul
- style de programmation : exemple précédent (`filter`)

Attention au coût du `try`

Exceptions en Java

Une exception est une rupture de calcul.
utilisée :

- pour éviter les erreur de calcul
  - division par zéro
  - accès à la référence `null`
  - ouverture d'un fichier inexistant
  - ...
- comme style de programmation

En Java une exception est un objet

Exceptions en Java : Syntaxe

- pose d'un récupérateur :

```
instruction ::=
  try {instructions}
  catch (type variable) { instructions}
  ...
  finally { instructions }
```

- déclenchement :

```
  throw ThrowableObject ;
```

- indication d'échappement d'une méthode

```
throws ExceptionClass
```

Exemples d’exceptions (1)

```
class Exemple1 {
  public int division(int a,int b)  {
      return (a/b);
  }
}

class Division_par_zero extends Exception {
}
```

Exemples d'exceptions (2)

```
class Exemple2 {

  private int division_aux(int a,int b)
  throws Division_par_zero
  {   if (b == 0)
      throw new Division_par_zero();
      else return (a/b);
  }


  public int division(int a, int b)
  {
    try { return division_aux(a,b); }
    catch(Division_par_zero e) { return 0;}
    finally {System.out.println("on passe ici");}
  }
}
```

La clause `finally` est facultative. Son code est exécuté à la sortie du `try`.

Etiquettes dynamiques en C

Pour certaines applications l'exécution du programme doit être reprise en un point particulier.

- Pour cela il est nécessaire d'utiliser des étiquettes dynamiques.
- On mémorise l'état de la pile d'exécution du processus dans une variable de type `jmp_buf` défini dans `<setjmp.h>` avec la fonction `setjmp`.
- Cette variable constitue l'étiquette. La fonction `longjmp` permet ensuite de reprendre l'exécution au point prévu en restaurant l'état de la pile.

```
int setjmp(jmp_buf env);
void longjmp (jmp_buf env, int val);
```

Exemple : produit d'une liste

```
int mult_liste_v2(jmp_buf env,liste_entier l) {
  compteur++;
  if (l == NULL) return 1;
  else { if (l->car == 0) longjmp(env,1);
         else return l->car*mult_liste_v2(env,l->cdr);
       }
}
int mult_liste(liste_entier l) {
  int r;
  jmp_buf env;
  compteur=0;
  switch(setjmp(env)){
    case 0: { r=mult_liste_v2(env,l);
              printf("calcul en %d etapes : ",compteur);
              return r;
            }
    case 1: { printf("un zero rencontr\'e \'a l'\'etape %d : ",compteur);
              return 0;
            }
    default: {fprintf(stderr,"cas incroyable\n");exit(0);}
  }
}
```

Exceptions en C (1)

Le mécanisme d'étiquettes dynamiques permet de construire un mécanisme d'exceptions (comme en ML ou ADA).

```
include "exception.h"

exception e;

Test()
{
  TRY
    Body();
  EXCEPT(e)
    Handler();
  ENDTRY
}
```

avec le déclenchement d'une exception par :

```
RAISE(e,v)
```

où e est une exception et v un entier.

Exceptions en C (2)

```
#include <setjmp.h>
typedef  char * exception;

typedef struct _ctx_block {
  jmp_buf env;
  exception exn;
  int val;
  int state;
  int found;
  struct _ctx_block *next;
} context_block;

#define  ES_EvalBody 0
#define  ES_Exception 1

extern exception ANY;
extern context_block *exceptionStack;
extern void _RaiseException();

#define RAISE(e,v) _RaiseException(&e,v)
```

Exceptions en C (3)

```
#define TRY \
  {\
    context_block _cb;\
    int state = ES_EvalBody;\
    _cb.next=exceptionStack;\
    _cb.found=0;\
    exceptionStack=&_cb;\
    if (setjmp(_cb.env) != 0) state=ES_Exception;\
    while(1){\
      if (state == ES_EvalBody){

#define EXCEPT(e)\
    if (state == ES_EvalBody) exceptionStack=exceptionStack->next;\
    break;\
    }\
    if (state == ES_Exception) \
      if ((_cb.exn == (exception)&e) || (&e == &ANY)) {\
        int exception_value = _cb.val;\
        _cb.found=1;
```

Exceptions en C (4)

```
#define ENDTRY \
   }\
   if (state == ES_EvalBody) {exceptionStack=exceptionStack->next;break;}\
   else {exceptionStack=exceptionStack->next;\
         if (_cb.found == 0) _RaiseException(_cb.exn,_cb.val); else break;}\
   }\
}
```

référence : Eric S. Robert, "Implementing Exceptions in C", Rapport de recherche SRC-40, Digital Equipment, 1989.

Exceptions en C : fichier C (5)

```
#include <stdio.h>
#include "exception.h"

context_block *exceptionStack = NULL;

exception ANY;

void _RaiseException(exception e, int v)
{
  if (exceptionStack == NULL)
  {fprintf(stderr,"Uncaught exception\n");
   exit(0);
  }
  else {
   exceptionStack->val=v;
   exceptionStack->exn=e;
   longjmp(exceptionStack->env,ES_Exception);
  }
}
```

Exceptions en C : exemple

```
#include "exception.h"
exception Found_zero;
int mult_liste_v3(liste_entier l){
 compteur++;
 if (l == NULL) return 1;
 else { if (l->car == 0) RAISE(Found_zero,1);
        else return l->car*mult_liste_v3(l->cdr);
      }
}
int mult_liste(liste_entier l){
  int r;  compteur=0;
  TRY
    { printf("calcul en %d etapes : ",compteur);
      r=mult_liste_v3(l);
    }
  EXCEPT(Found_zero)
    { printf("un zero rencontr\'e \'a l'\'etape %d : ",compteur);
      r=0;
    }
  ENDTRY
  return r;
}
```

Continuations

Une continuation est la représentation d'un contexte de calcul sous la forme d'une fonction.

- utilisées pour décrire les ruptures de contrôle dans les formalismes de définition de la sémantique des langages de programmation.
- popularisées par le langage Scheme
- à la base d'un style de programmation appelé CPS (*Continuation Passing Style*) qui permet d'expliciter le contrôle.

Style CPS

La transformation d'une fonction en style CPS se fait en lui ajoutant un argument supplémentaire (la continuation initiale), et en explicitant sous la forme d'une continuation le contexte d'évaluation de chaque calcul intermédiaire.
la fonction fib :

```
let rec fib(n) =
  if n <= 1 then 1 else
  fib(n-1) + fib(n-2)
```

devient

```
let rec fib(n)(cont) =
 if n <= 1 then cont(1) else
 fib(n-1)
    (fun m1 ->
       fib(n-2) (fun m2 -> cont(m1+m2)))
```

et les appels

```
fib(5)(fun x -> x);;
fib(5)(print);;
```

continuation courante (1)

Certains langages (Scheme, SML/NJ) fournissent une primitive permettant de capturer le contexte courant d'évaluation sous la forme d'une fonction appelée la *continuation courante*, de la lier à un identificateur, et de l'utiliser si nécessaire.

- Cette continuation, lorsqu'elle est utilisée dans un certain contexte, ignore ce contexte et restaure le contexte qu'elle représente.

continuation courante (2)

- En Scheme, la capture de la continuation courante est effectuée par la primitive `call-with-current-continuation`, ou, plus simplement, `call/cc`, qui reçoit une fonction à un paramètre (`fun k -> b`) et qui l'applique à la continuation courante.
- Le contrôle est donc passé au corps `b` de la fonction, où la continuation capturée est disponible sous le nom `k` du paramètre formel.

Ainsi, la valeur de:

```
call/cc
 (fun throw ->  f (if p then throw(0) else ...))
```

sera `0` si `p` vaut `true`

En Scheme : call/cc (1)

Exemple : produit des éléments d'une liste (classique)

```
(define (list_mult_aux return l)
  (letrec ((r (lambda (l) (print l)
              (if (null? l) 1
              (if (= (car l) 0) (return 0)
               (print(* (car l) (r (cdr l))))))))))
     (r l)))

(define (list_mult l)
   (call/cc (lambda (c) ((list_mult_aux c l)))))

(list_mult '( 1 2 3))
```

En Scheme : call/cc (2)

Exemple : produit des éléments d'une liste (classique)

```
(list_mult '( 1 2 3 0 4 5))
(1 2 3 0 4 5)
(2 3 0 4 5)
(3 0 4 5)
(0 4 5)
0
```

En Scheme : call/cc (3)

Exemple : produit des éléments d'une liste (call/cc)

```
(define (main x)
    (+ (call/cc (lambda (k0) (f1 k0 x))) 3) )

(define (f1 k0 x)
    (+ (call/cc (lambda (k1) (f2 k0 k1 x))) 2) )

(define (f2 k0 k1  x)
    (+ (call/cc (lambda (k2) (f3 k0 k1 k2 x))) 1) )

(define (f3 k0 k1 k2 x)
   (cond ((< x 10) (k2 x))
         ((< x 100) (k1 x))
         (#t  (k0 x))))
```

En Scheme : call/cc (4)

Exemple : produit des éléments d'une liste (call/cc)

```
(main 0)
6

(main 10)
15

(main 100)
103
```

Le call/cc typé (1)

- Exemple 1 :

```
     (print x);  (* paramètre inutile (unit) *)
---> point de calcul
     ...
```

- Exemple 2 :

```
(3+4*2)     paramètre indispensable
     |
     |
 point de calcul
```

On voit bien que sans paramètre pour remplacer la valeur 2 on ne peut pas continuer le calcul

Le call/cc typé (2)

- Le call/cc applique à une fonction passée en paramètre la continuation courante.
- Le type $((' a \Rightarrow' a) \Rightarrow' a) \Rightarrow' a$
- Exemple :

```
(3+4*(call_cc
(fun k ->
   10*
   (if read_bool()
    then (k 3)
    else 4))))
```

Implantation du call/cc (1)

- Implémentation classique : le CPS
  - Solution élégante et « rapide »
  - Ralentit les programmes qui ne l'utilise pas
  - Interfaçage avec C difficile

référence : Juliusz Chroboczek. Continuation Passing for C: A space-efficient implementation of concurrency

Implantation du call/cc (2)

Machines à pile: :

- Qu’est-ce qu’un point de calcul ?
  - Un contexte d’éxécution
  - Une copie de pile
  - Un tas partagé
- Implémentation lourde
  - Interfaçage avec C plus facile
  - Coût plus «juste».
  - Pb de vitesse

Le call/cc utilisation

- Exception (sous cas du call/cc)
- interprète : Luc Moreau, Daniel Ribbens, and Pascal Gribomont. Advanced Programming Techniques Using Scheme. In Journées Francophones des Languages Applicatifs
- Nagivateur : Christian Queinnec, The Influence of Browsers on Evaluators or, Continuations to Program Web Servers,icfp2000
- Programmes concurrents : Luc Moreau. Continuing into the Future: the Return, InterSymp'96, Luc Moreau. The Semantics of Scheme with Future.ICFP'96.
- Migrations de calcul: Chailloux - Ravet - Verlaguet : hirondML : http://www-spiral.lip6.fr/~chaillou/Public/Dev/HirondML/